This paper appeared in Proc. of IEEECS Annual Symposium on VLSI (ISVLSI 2003), Tampa, Florida, February 20-21, 2003.

Using Dynamic Branch Behavior for Power-Efficient Instruction Fetch

J. S. Hu, N. Vijaykrishnan, M. J. Irwin, M. Kandemir
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{jhu,vijay,mji,kandemir}@cse.psu.edu

Abstract

Power consumption has become an increasing concern in high performance microprocessor design in terms of packaging and cooling cost. The fetch unit including instruction cache contributes a large portion of the total power consumption in the microprocessor. The instruction cache itself suffers some hidden power consumption due to dynamic control flows. Although capturing the dynamic control flows to boost performance, conventional trace caches (CTC) may increase power consumption in the fetch unit due to its simultaneous access to both the trace cache and the instruction cache. By avoiding this simultaneous accesses, sequential trace caches (STC) achieve lower power consumption, but suffer a significant performance loss at the meantime. In this paper, we propose dynamic direction prediction based trace cache (DPTC), which avoids simultaneous accesses to the trace cache and the instruction cache with the guide of fetch direction prediction. Experimental results show that dynamic prediction based trace cache can achieve 38.5% power reduction over conventional trace caches and an additional 7.2% reduction over STC, on average, while only trading a 1.8% performance loss compared to CTC.

1. Introduction

Approaching the upper bound of available performance with the advances in technology has been the main goal in high performance microprocessor designs. Fetch unit is becoming one of the main performance bottlenecks as the issue width and number of function units continue to increase [1]. Providing a higher fetch bandwidth by applying new technologies such as branch address cache (BAC) [2], collapsing buffer (CB) [3] and trace cache [4] is the key to performance advancement. At the meantime, power consumption has become an increasing concern in high performance microprocessor design in terms of packaging and cooling cost. Note that power optimizations should only have small impact on performance in this context. In [5], it indicates that the instruction cache contributes a large portion of the total power consumption in typical microprocessors. One recent work elaborates that a 32KB L1 instruction cache consumes 22% of the total power in a system with disk [6]. Thus optimizing the power consumption in the fetch unit is particularly important.

In the instruction cache, each cache line stores instructions with the static order generated by the compiler. At any given fetch cycle, at most one cache line (non-interleaved cache structure) is fetched. In traditional fetch mechanisms, instructions after the first branch (single-block fetching) or the first taken branch (multi-block fetching) within a cache line are discarded. These mechanisms not only waste the fetch bandwidth of the instruction cache, but also waste power in fetching the whole cache line. It may also increase the power consumption due to the increased access transactions to the instruction cache and increased execution time. In our previous work [7], a detailed scenario was introduced to elaborate this hidden power consumption in the instruction cache. We found that the main reason for this problem is that the dynamic sequence of instructions at run time is quite different from the static sequence which is stored in the instruction cache.

Thus some new mechanisms that can utilize the dynamic characteristics of instructions must be introduced to alleviate this problem. We observe that the trace cache [4] might be a good candidate to improve the power efficiency in the fetch unit due to the dynamic sequences stored in the trace cache. The microarchitecture of a conventional trace cache (CTC) is given in Figure 1. In conventional trace cache mecha-

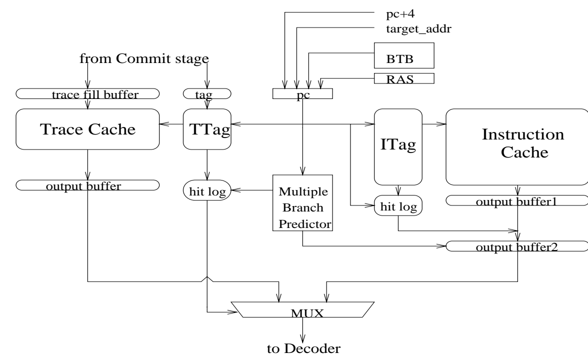


Figure 1. Microarchitecture of conventional trace cache.

nism, the trace cache and L1 instruction cache (L1 I-Cache) are accessed simultaneously to reduce the miss penalty from the trace cache. A hit in the trace cache aborts the transaction in L1 I-Cache. Otherwise, it proceeds just as the traditional I-Cache fetch mechanism. Normally, the conventional trace cache can deliver a moderate performance improvement over multi-block fetching I-Cache. Our results [8] show that some configurations of conventional trace cache do increase the power consumption in the fetch unit. The main reason for this power increase is the concurrent access to both trace cache and L1 I-Cache.

In [9], a sequential trace cache (STC) is investigated for its impact on the fetch width. We observe that the sequential scheme implies the potential to reduce the power consumption in the fetch unit. Our previous work [7] provides a more detailed power/performance analysis of the sequential trace cache and includes a comparison with both the conventional trace cache and instruction cache. It shows sequential trace caches can reduce the fetch power by 32% on average, but suffer an average 12% performance loss while compared to conventional trace cache. Selective trace cache (SLTC) proposed in [7] has the superiority over conventional trace cache and sequential trace cache in terms of a high performance close to CTC and a lower power consumption than STC. It uses both compiler optimization and hardware support to selectively control trace cache lookups and updates. On the other hand, SLTC is profile based and needs an additional training process as well as a modification in ISA (Instruction Set Architecture).

In this paper, we propose a pure hardware scheme to implement selective fetch between the trace cache and instruction cache: *dynamic direction prediction based trace cache (DPTC)*. DPTC augments the function of existing BTB (Branch Target Buffer) to incorporate a fetch direction predictor (FDP). At branch prediction, a fetch direction prediction is carried out by the FDP. If FDP predicts that next trace will hit the trace cache, only the trace cache will be accessed in the next fetch cycle. Otherwise, only the instruction cache is accessed. Any miss in the trace cache incurs one cycle penalty and the next fetch will be directed to the instruction cache. This predictor captures the locality of trace accesses and deliver very higher prediction accuracy. Our experimental results show that DPTC can reduce the fetch power in the fetch unit by 38.5% over CTC and additional 7.2% over STC, an average for all configurations and benchmarks, while sacrificing only 1.8% performance (instructions per cycle) loss as compared to CTC.

The rest part of this paper is organized as follows. In Section 2, we propose our dynamic direction prediction based trace cache. The experimental model is detailed in Section 3. In Section 4, we present the experimental results. We discuss some related work in Section 5. Section 6 concludes our work.

2. Dynamic Direction Prediction based Trace Cache

Profile-based selective trace cache works very well in terms of high performance and low power consumption in the fetch unit. The key point of selective trace cache is that it has precomputed (by compiler) information of the dominant trace set, which captures the locality of trace accesses [7]. This information is used to control the fetch unit only to access the trace cache or the instruction cache at any given fetch stage. A significantly improved trace cache hit rate and reduced trace cache lookups (resulting in misses) help achieve the high performance and low power consumption of selective trace cache. This benefit comes at the cost of ISA modification and the additional profiling time.

We observe that the locality of trace cache accesses is only another representative of the locality of the original code during execution. This locality can be also effectively captured and utilized by pure dynamic schemes. In this section, we propose our dynamic direction prediction based trace cache (DPTC), which is powered by a dynamic fetch direction predictor (FDP). FDP captures the history information and locality of trace cache accesses. It predicts whether the next trace will hit the trace cache or not based on history information collected at runtime. This prediction controls the fetch unit to only access the trace cache or the instruction cache in the next fetch cycle. Working in this way, the concept of selective fetch is purely implemented in hardware scheme.

2.1. Dynamic Fetch Direction Predictor

The dynamic fetch direction predictor is the key part of the direction prediction based trace cache. An effective fetch direction predictor must satisfy the following two requirements.

- Maintaining the capability of trace cache to supply effective instructions. If the constraints on the fetch unit, put by the direction predictor, restrain the trace cache from supplying instructions on the correct execution path, the argument of using such a predictor will be weakened. This capability is measured as the ratio of committed instructions supplied by the trace cache. It is one of the main metrics for the effectiveness of the trace caches.
- High prediction accuracy. The prediction accuracy is defined as the ratio between the actual number of trace cache hits to the number of predicted trace cache hits. Note that trace cache is only accessed when the trace is predicted as hit in the previous fetch cycle. This requirement also guarantees a controlled performance penalty due to trace cache misses. High prediction accuracy also reduces the unnecessary accesses (miss the trace cache) to the trace cache thus reducing the power consumption in the fetch unit.

We incorporate our dynamic direction predictor using the existing branch predictor to ensure that the prediction happens one cycle ahead of the next fetch cycle. The BTB (Branch Target Buffer) of the branch predictor is extended in the following two ways. First, both taken and not-taken branches can update their BTB entries. This is in contrast to the base configuration in which only a taken branch updates its BTB entry in the commit stage. Second, the BTB entry is

expanded to have two additional saturating counters for dynamic direction predictor. The first counter (*t_cnt*) is used to predict whether the target address leads to a trace currently in the trace cache, the other one (*f_cnt*) for predicting whether the fall-through address leads to a trace existing in the trace cache. Both of them are two bit saturating counters. The new structure of the extended BTB entry is given in Figure 2.

Figure 2. The structure of extended BTB entry (A tagged BTB is used in this paper).

The update of these two counters (*t_cnt* and *f_cnt*) is performed according to the automaton given in Figure 3. The direction predictor is updated at the commit stage. Each time when a new BTB entry is created, the states of these two counters are set to $A1(00)$. Note that the target address is updated only when the first taken occurs. When a new trace is updated to the trace cache, the corresponding counter t_cnt/f_cnt in the BTB entry of the previous branch, whose target/fall-through instruction leads to the creation of this new trace, is set to $A3(11)$. Trace cache hit/miss information integrated with the branch update information is used to update the counter state. Any trace cache hit will increase the counter upwards state $A3$, and trace cache miss will decrease the counter downwards $A1$. If the counter is in state $A2(10)$ or $A3(11)$, the predictor predicts that the next trace will hit the trace cache and gives the prediction "access to the trace cache (TC)". Otherwise, a prediction of "access to the instruction cache (IC)" is given.

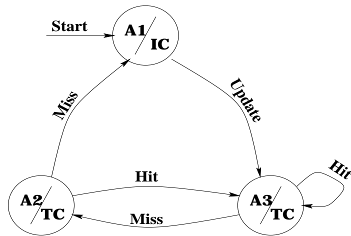


Figure 3. State diagram of the finite-state Moore machine for fetch direction prediction and counter update.

Since the trace hit/miss information is embedded into the branch update information, only a slight modification is needed to extract this information to update the fetch direction predictor. On the other hand, when a new trace is written into the trace cache, the branch leading to this trace has already been retired from the datapath. Although the trace fill-unit keeps the latest branches (up to 3 branches) within the current trace, the information (PC address and outcome) of the branch (leading to the new built trace) is not available at this time. We introduce a tiny branch buffer with four entries to continuously keep the last four committed branches. This buffer structure is incorporated with the commit unit. The branch buffer, which is given in Figure 4, is maintained as a simple circular queue with a 2-bit head pointer pointing to the slot for the next branch to be committed.

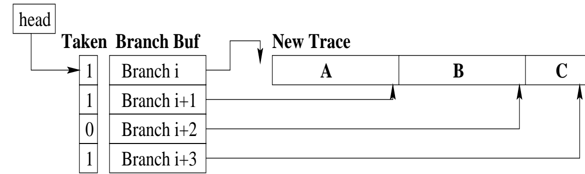


Figure 4. The structure of the branch buffer. It shows the snapshot of the branch buffer content when a new trace with three branches finishes its building.

The branch buffer keeps the PC address of the last four branches committed as well as its outcomes. The outcome (taken field) of each branch uses 1 bit. A value of "1" means the branch is taken, and "0" for non-taken outcome. Since a trace contains at most three branches (Note that the tag filed of trace keeps the number of branches in current trace), keeping the last four branches committed is sufficient to figure out the branch leading to the currently built trace. As the snapshot shown in Figure 4, the branch in slot 0 is the one leading to current trace ABC. When the new trace is updated to the trace cache, the fetch direction predictor is also updated using this branch address and its outcome.

2.2. Microarchitecture of DPTC

The microarchitecture of the dynamic direction prediction based trace cache is given in Figure 5. It uses an additional machine flag (direction flag) to control the access to the trace cache or the instruction cache. This direction flag is set by the outcome of the fetch direction predictor.

Since the fetch direction predictor does not keep information about the branches within each trace, supporting partial matching [10] helps reduce the one cycle performance penalty due to a misprediction. Our previous work [7] shows that partial matching has little impact on power consumption in the fetch unit.

Dynamic direction predictor based trace cache works in the following ways. Note that a branch predictor supporting at most three predictions per cycle is used throughout this paper. At a given fetch cycle, if the last effective branch (the first predicted-taken branch, or the third not-taken branch, or the branch instruction ending a cache line) in a fetched instruction cache line or the branch instruction ending a fetched trace is predicted taken (not taken), the counter *t_cnt* (*f_cnt*) within its BTB entry is checked. If the counter is in states $A2(10)$ or $A3(11)$ (only the higher bit is checked), a prediction of "access to the trace cache (TC)" is signaled (direction flag, shown in Figure 5 is set to 1), and only the trace cache is accessed in the next fetch cycle. If the counter is at states

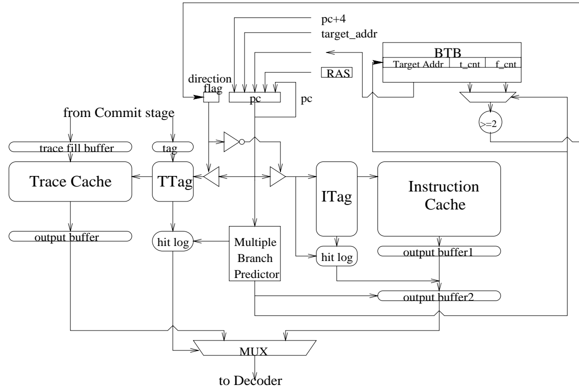


Figure 5. Microarchitecture of the dynamic direction prediction based trace cache.

$A1(00)$ (the higher bit is "0"), or if there was a BTB miss, a prediction of "access to the instruction cache (IC)" is given (direction flag is set to 0), and only the instruction cache is accessed in the next fetch cycle.

As the fetch direction predictor only adds 4 bits to each BTB entry, and the direction prediction circuit is very simple (only one bit from each counter is checked), the power consumption for direction prediction is expected to be very small. Note that direction prediction happens at the same time as branch prediction. Thus no additional BTB access is required for direction prediction. Due to the very infrequent trace update (the rate of trace updates to BTB access is around 2.9%), the power consumption of direction predictor update for a new trace is negligible.

3. Experimental Model

We augmented Wattch [11], a power evaluation tool for superscalar processors, to model the different trace cache schemes proposed in this work. The power model of trace caches implemented is similar to the one in Wattch for the instruction cache. The power consumption of trace caches consist of two part: power consumed for trace cache lookup and power consumed for trace cache update. We define *fetch power* as the power consumed in both L1 instruction cache and trace cache.

The simulated processor has a datapath width of sixteen instructions, supporting out-of-order issuing. The instruction window size is 256 and the load/store queue size is 128. The configuration of the memory hierarchy is given in Table 1.

We use a modified gshare [12] predictor that is capable of making three branch predictions per cycle to support multiblock fetching and trace cache. It has a 16 bit long branch history register. A 32-entry return address stack and a 4-way 1024 set (total 4096 entries) branch target buffer are used to generate predicted target address.

Throughout this paper, we use following configurations for fetch unit, which are listed in Table 2.

Memory Hierarchy	Configuration
Trace Cache	8-64KB, 2-way, 128B block, 1 cycle
L1 ICache	8-64KB, 2-way, 128B block, 1 cycle
L1 DCache	64KB, 4-way, 64B block, 1 cycle
L2 UCache	512KB, 4-way, 128B block, 6 cycles
Memory	30 cycles for first chunk, 4 cycles rest
TLB	4-way 64-entry ITLB, 4-way 128-entry DTLB 30 cycles to service a TLB miss

Table 1. Configuration of memory hierarchy.

Configuration	Meaning
CTC-8k	Seq3 (64KB L1 ICache) + 8KB conventional (conv.) trace cache
CTC-32K	Seq3 (32KB L1 ICache) + 32KB conv. trace cache
CTC-64K	Seq3 (8KB L1 ICache) + 64KB conv. trace cache
STC-8K	Seq3 (64KB L1 ICache) + 8KB sequential (seq.) trace cache
STC-32K	Seq3 (32KB L1 ICache) + 32KB seq. trace cache
STC-64K	Seq3 (8KB L1 ICache) + 64KB seq. trace cache
DPTC-8K	seq3 (64KB L1 ICache) + 8KB DPTC
DPTC-32K	seq3 (32KB L1 ICache) + 32KB DPTC
DPTC-64K	seq3 (8KB L1 ICache) + 64KB DPTC

Table 2. Cache configurations.

Ten integer benchmarks from SPEC2000 CINT are selected in this experiment using their *reference* inputs. All benchmarks except *bzip2* and *mcf* are first fast forwarded 300 million instructions, then simulated 200 million instructions in following experiments. No instruction is fast forwarded for *bzip2* and *mcf* due to their specific characteristics [13].

4. Experimental Results

Dynamic fetch direction predictor is the key part in DPTC. The performance and power efficiency of DPTC are directly determined by the effectiveness of this predictor. The direction predictor's effectiveness comes from two part: ratio of committed instruction supplied by the trace cache and prediction accuracy, as discussed in Section 3.1. The ratio is defined as the percentage of committed instructions from the trace cache to the total number of committed instructions. The direction predictor should not degrade this ratio. Prediction accuracy is defined as the ratio between the number of actual trace cache hits to the number of predicted hits which is equal to the number of trace cache lookups in DPTC. Prediction accuracy is also equal to the trace cache hit rate in DPTC.

The ratio of committed instructions supplied by the trace cache shows the effectiveness of trace cache to supply instructions on the correct paths. The results given in Table 3 illustrate that the fetch direction predictor in DPTC does not degrade the ability to supply effective instructions for SPEC2000 CINT benchmarks.

The direction prediction accuracy is critical to the performance and fetch power consumption in DPTC. A high prediction accuracy reduces the one cycle performance penalty due to trace cache misses. It also helps reduce the power consumption by decreasing the number of trace cache lookups resulting in misses. The prediction accuracy for three configurations is given in Figure 6. Especially for benchmark

Conf./Commit Ratio	CTC	STC	DPTC
IC-64K.TC-8K	0.4511	0.4419	0.4531
IC-32K.TC-32K	0.5497	0.5282	0.5429
IC-8K.TC-64K	0.5714	0.5440	0.5627

Table 3. The average percentage of committed instructions supplied by trace caches for ten SPEC2000 CINT benchmarks at three configurations.

Conf./Hit Rate	CTC	STC	DPTC
IC-64K.TC-8K	0.3804	0.3920	0.7745
IC-32K.TC-32K	0.4890	0.4908	0.9234
IC-8K.TC-64K	0.5045	0.5162	0.9535

Table 4. The average trace cache hit rate for ten SPEC2000 CINT benchmarks at three configurations.

bzip2, its prediction accuracy is more than 99%. On average, the prediction accuracy is 77.4% for DPTC-8K, 92.3% for DPTC-32K, and 95.3% for DPTC-64K. In contrast to this high prediction accuracy (trace cache hit rate) in DPTC, conventional trace cache and sequential trace cache have a lower hit rate in the trace cache. The average hit rate comparison of these three trace caches at different configurations is given in Table 4.

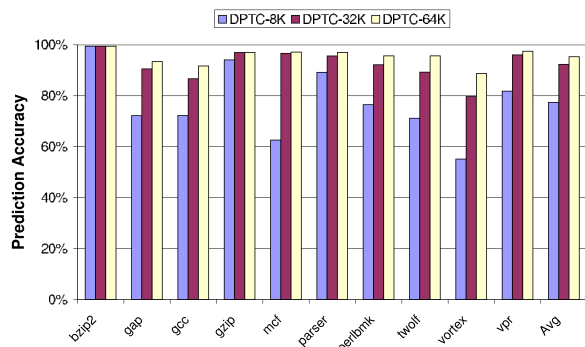


Figure 6. Prediction accuracy of the dynamic direction predictor at three configurations.

Maintaining the effectiveness of supplying instructions on the correct execution paths and a high prediction accuracy, dynamic direction prediction based trace cache (DPTC) presents a very nice combination of high performance and low power consumption in the fetch unit. From Figure 7 (a), a small performance degradation is observed, an average of 1.8% (for three configurations) compared to conventional trace cache. At the same time, due to its high prediction accuracy and sequential nature, DPTC also achieves a significant power reduction in the fetch unit, which is shown in Figure 7 (b), an average of 38.5%. It also reduces the fetch power by 7.2%, on average, over the sequential trace caches.

The results presented in Figure 7 confirms that the hardware prediction scheme for fetch control in DPTC competes the profile-based compiler scheme in selective trace cache [7] in terms of high performance and lower power consumption. On the other hand, DPTC has several advantages over selective trace cache such as low cost in implementation, application independence, and platform compatibility.

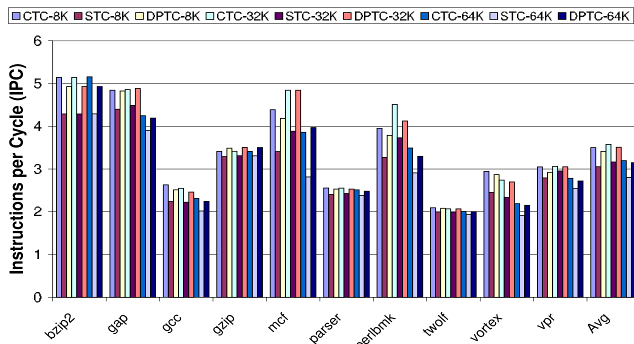
5. Related Work

There has been a lot of previous research work on improving the performance of conventional trace caches [10][14][15]. Friendly et al. [10] proposed two techniques, partial matching and inactive issue. Another two techniques, branch promotion and trace packing were examined by Patel et al. [14]. The goal of these techniques is to increase the instruction fetch rate thus to improve the performance of trace caches. Rotenberg et al. [15] presented a trace cache microarchitecture using trace-level sequencing and next trace prediction to deliver higher performance over multiple-block fetch mechanism. In contrast to their focuses on the performance improvement of conventional trace caches, our work is to develop a new microarchitecture for trace caches that achieve a significant reduction in fetch power while maintaining high performance.

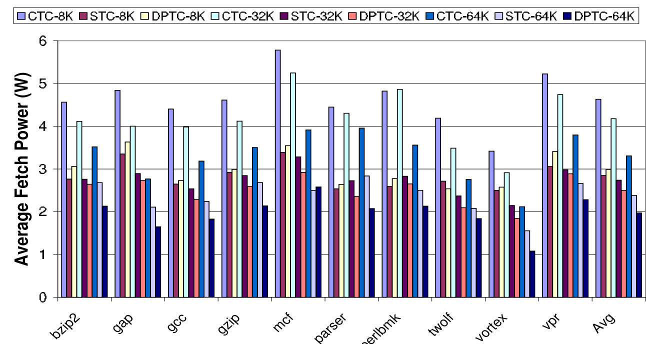
Rosner et al. [16] propose their filtering techniques to reduce the trace build thus to decrease power consumption and increase overall performance. Our work exploits the selective fetch in the fetch unit of DPTC to reduce the power consumption in terms of fetch power. Reordering the code to improve sequentiality as employed in software trace cache [17] is another alternative to hardware trace caches. The software trace cache combined with a hardware trace cache is often attractive for performance. Profile based selective trace cache [7] uses a profiling process to compute the trace dominant set and selectively controls the access to trace cache or instruction cache and the trace build according to the dominant set. Our current work differs from these two work in that 1) it is independent of compiler optimizations or code layout optimizations, 2) it provides a pure hardware scheme to implement this selective fetch rather than profile-based software schemes, 3) it avoids any impact on the current ISA architecture which makes it independent of the underlying platforms.

6. Conclusions

Power consumption in high performance microprocessors is becoming an important issue for the state-of-the-art designs. Achieving a significant power reduction while maintaining the original high performance has become one of the main principles for designs in this context. This paper explores the potential to reduce the power consumption in the



(a) Performance comparison.



(b) Power comparison.

Figure 7. Performance and power comparison among conventional trace cache (CTC), sequential trace cache (STC), and dynamic direction prediction based trace cache (DPTC).

fetch unit with a high performance microprocessor infrastructure, conventional trace cache. Sequential trace cache trades a considerable performance loss for a significant fetch power reduction. By selectively control the fetch unit to only access the trace cache or the instruction cache, profile based selective trace cache has been shown its superiority in terms of high performance close to conventional trace cache and lower power consumption than sequential trace cache. In this paper, we propose a dynamic direction prediction based trace cache, in which the fetch direction predictor gives prediction whether next trace will hit the trace cache or not. This prediction controls the fetch unit to direction the fetch address only to the trace cache or the instruction cache. Dynamic direction prediction based trace cache is a pure hardware implementation. As the design of the fetch direction predictor achieves a high prediction accuracy, our DPTC achieves similar effectiveness in terms of performance and low fetch power as to the selective trace cache while avoiding any additional profiling process, recompiling, and ISA modification that would be needed in selective trace cache.

Acknowledgments

This work was supported in part by a grant from MARCO 98-DF-600 GSRC, NSF CAREER Awards 0093085, and NSF Awards 0082064 and 0103583.

References

[1] G. Hinton *et al.*, "The microarchitecture of the pentium 4 processor," *Intel Technical Journal*, vol. Q1, Feb. 2001.

[2] T. Y. Yeh, D. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proc. of the 7th Intl. Conference on Supercomputing (ICS'7)*, July 1993.

[3] T. Conte and the others, "Optimization of instruction fetch mechanism for high issue rates," in *Proc. of the 22th ISCA* , June 1995.

[4] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proc. of the 29th MICRO*, November 1996.

[5] J. Montanaro and et al, "A 160-mhz, 32-b, 0.5-w cmos risc microprocessor," *Digital Technical Journal, Digital Equipment Corporation*, vol. 9, 1997.

[6] S. Gurumurthi, A. Sivasubramaniam, M. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. John, "Using complete machine simulation for software power estimation: The softwatt approach," in *Proc. of the 8th HPCA*, February 2002.

[7] J. S. Hu, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir, "Selective trace cache: A low power and high performance fetch mechanism," Tech. Rep. CSE-02-016, Pennsylvania State University, 2002.

[8] J. S. Hu, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Power-efficient trace caches," in *Proc. of the 5th Design Automation and Test in Europe Conference (DATE'02)*, March 2002.

[9] J. Faistl and T. Jaracz, "Trace cache: Effect on instruction cache miss frequency." http://www.ece.cmu.edu/∼ee742/proj_s98/faistl/index.html, 1998.

[10] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative fetch and issue policies for the trace cache fetch mechanism," in *Proc. of the 30th MICRO*, December 1997.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. of the 27th ISCA*, June 2000.

[12] T. Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proc. of the 20th ISCA* , May 1993.

[13] R. Cooksey and D. Grunwald, "Characterization of the spec2000 benchmark suite." http://www.cs.colorado.edu/∼rcooksey/pubs.html, 2001.

[14] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proc. of the 25th ISCA*, June 1998.

[15] E. Rotenberg, S. Bennett, and J. Smith, "A trace cache microarchitecture and evaluation," *IEEE Transactions on Computers (special issue on cache memory)*, vol. 48, pp. 111–120, February 1999.

[16] R. Rosner, A. Mendelson, and R. Ronen, "Filtering techniques to improve trace-cache efficiency," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[17] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *Proceedings of the 13th Intl. Conference on Supercomputing*, June 1999.